Raunak Narwal
Department of Mathematical Sciences
Indian Institute of Science Education and Research, Mohali, 130406,
Punjab

December 12

Introduction

We have added comments, docstrings and created UML diagrams for all of our scripts. UML files were created using planttext website, it take PUML files and outputs PNGs and SVGs. I have added Numpy styled Docstrings using Pyment library, afterwards i have manually edited them to make them more descriptive and useful. I would try to include all the basics that build up to make our code. The information in the scripts, Readme file and this report overlaps. Each of them has their own purpose and is useful in its own way.

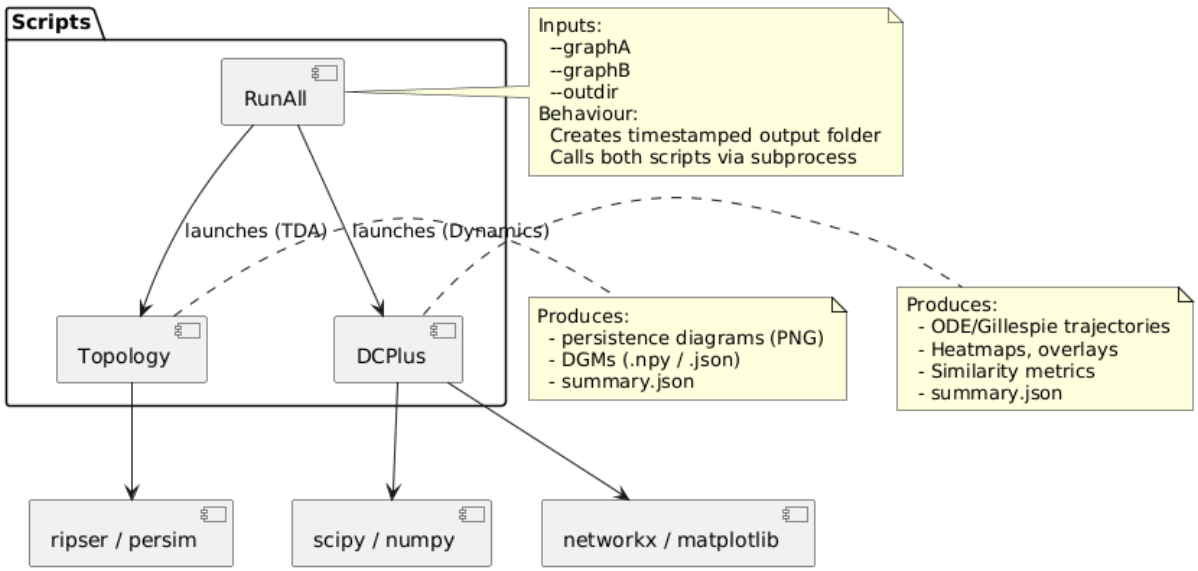

Figure 1: UML Diagram

Topology_Compare.py

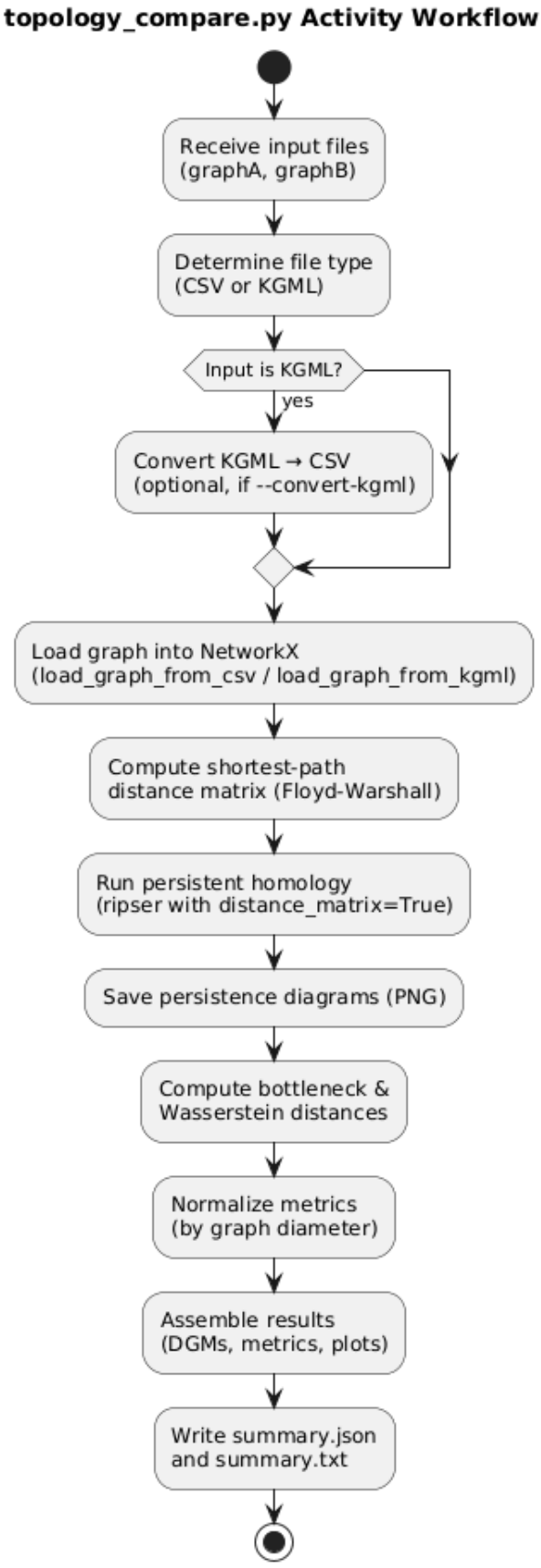

Figure 2: UML Diagram for Topology_Compare.py

Lines of Code Explained
The code imports argparse (for command line interface)
os/Pathlib for file and directory handling
json for saving metrics
numpy and pandas for data handling
networkx for graph operations
floyd warshall is useful in computing shortest path matrix
ripser is imported for persistent homology computation
persim for plotting and distances
seaborn for aesthethic graphs and plotting
xml.etree.ElementTree for parsing XML files

ensure_outdir(path)
This function checks if the output directory exists, if not it creates one.
It wraps a string path into a path object
calls mkdir method with exist_ok = True, it creates nested folders if needed and prevents errors if directory alreadt exists.

kgml_to_csv(kgml_path, out_csv)
This function converts pathways from KEGG KGML format into CSV format so that rest of the pipeline can easily read.
KGML contains entry notes : species , enzymes and compunds
relation edges : regulation links and reaction edges : metabolic conversions
The function works by parsing the XML tree structure of KGML file

```
tree = ET.parse(kgml_path)
root = tree.getroot()
```

It iterates over entries and relations to extract nodes and edges

```
    entries[eid] = name
    edges.append((source_name, target_name, rtype))
```

Extracts reactions

```
<reaction>
   <substrate id="X"/>
   <product id="Y"/>
</reaction>
```

the loop

```
        for s in substrates:
        for p in products:
            edges.append((entries[s], entries[p], "reaction"))
```

Then it converts them into pandas DataFrames and saves them as CSV files.

load_graph_from_csv(csv_path)
This function loads a CSV file into NetworkX graph

```
        df = pd.read_csv(csv_path)
        df.columns = [c.lower() for c in df.columns]
```

it reads the CSV using pandas, creates a directed graph internally
Then it builds an undirected projection, because distance matrix must be symmetric
Persistent Homology for asymmetric graphs is not well defined.

load_graph_from_kgml(kgml_path)
this is similar to above but it directly constructs a networkX graph without saving CSV. the previous function outputs a CSV edge list but this outputs a graph, so they share logic but serve different parts of our pipeline.

graph_diameter(G)
This function computes the diameter of a graph G
which is the longest shortest path between any two nodes
it handles trivial cases

```
    if G.number_of_nodes() == 0:
        return 0
```

if the graph is connected, then use NetworkX built in BFS diameter

```
        if nx.is_connected(G):
        try:
            return int(nx.diameter(G))
```

and if it fails , then fallback

```
D = dict(nx.all_pairs_shortest_path_length(G))
maxd = max(d for u in D for d in D[u].values())
```

We compute diameter because later, bottleneck/wasserstein are normalized by graph diameter to make metrics comparable across networks.

shortest_path_distance_matrix(G, disconnected_value=None)
this is a very critical function in our pipeline, it converts the graph into N*N distance matrix which is then used by Ripser for computing persistent homology.
Lists nodes and indexing

```
nodes = list(G.nodes())
idx = {node: i for i,node in enumerate(nodes)}
```

this creates consistent ordering of nodes
initializes distance matrix with disconnected value (infinity by default)

```
D = np.full((n, n), np.inf)
np.fill_diagonal(D, 0)
```

fill matrix with edge weights

```
for u, v, data in G.edges(data=True):
w = data.get("weight", 1.0)
D[i,j] = w
D[j,i] = w
```

floyd warshall algorithm is used to compute all pairs shortest paths
this computes all pairs shortest paths efficiently
replaces infinities with a large finite value

```
if disconnected_value is None:
     disconnected_value = 2 * max(finite_distances)
D[~np.isfinite(D)] = disconnected_value
```

persistence_from_distance(D, maxdim=1, thresh=None)
purpose is to run Ripser persistent homology on the distance matrix
it sets threshold for filtration.

```
thresh = 95th percentile of distances
```

calls ripser with distance matrix and parameters

```
res = ripser(D, maxdim=maxdim, thresh=thresh, distance_matrix=True)
```

it outputs the diagram in a list form

persistence_to_dict(dgms)
it converts diagrams into a dictionary format for easier storage and retrieval

```
 {
 "H0": [[birth, death], ...],
  "H1": [[birth, death], ...]
}
```

plot_and_save(diagrams, title, outdir, tag)
creates a PNG image of the persistence diagram.
plot_graph(G, outdir, name)
this ceates the graphs visualisations
compute layout

```
pos = nx.spring_layout(G, seed=42)
cmap = plt.get_cmap('viridis')
```

degree based node size: nodes with higher degree appear larger

degree based color

compute_distance_metrics(dgA, dgB, p=2)
this computes the TDA distances between two diagrams
it computes bottleneck and wasserstein distances for each homology dimension

```
bn = bottleneck(A, B)
ws = wasserstein(A, B, p=p)
```

compare_diagrams_full(diagA, diagB, p=2)
it computes distances across all homology dimensions present

```
maxdim = max(len(diagA), len(diagB))
```

loops across dimensions

```
for dim in range(maxdim):
  A = diagA[dim]
  B = diagB[dim]
  bn, ws = compute_distance_metrics(A, B)
  metrics[f"H{dim}_bottleneck"] = bn
  metrics[f"H{dim}_wasserstein"] = ws
```

run_pipeline(graphA, graphB, nameA, nameB, outdir, maxdim, p)
this function integrates every stage, this is exatly what run_all.py calls.
ensures output directory

```
out = ensure_outdir(outdir)
```

Compute distance matrices

```
DA, nodesA = shortest_path_distance_matrix(graphA)
DB, nodesB = shortest_path_distance_matrix(graphB)
```

These produce: DA: NxN distance matrix for Graph A, DB: NxN distance matrix for Graph B and nodesA and nodesB: node lists in same order as matrices
then computes persistence diagrams

```
dgmA, resA = persistence_from_distance(DA, maxdim=maxdim)
dgmB, resB = persistence_from_distance(DB, maxdim=maxdim)
```

saves persistence diagrams

```
np.save(f"{nameA}_dgms.npy", dgmA)
np.save(f"{nameB}_dgms.npy", dgmB)
```

converts diagrams to JSON

```
json.dump(persistence_to_dict(dgmA), f)
json.dump(persistence_to_dict(dgmB), f)
```

```
{
"H0": [[0, 1], [0, 2], ...],
"H1": [[1.2, 3.5], ...]
}
```

saves all visual diagrams

```
figA = plot_and_save(dgmA, title=nameA, tag=f"{nameA}_tda")
figB = plot_and_save(dgmB, title=nameB, tag=f"{nameB}_tda")
```

compute raw topological distances
Normalize distances by graph diameter

```
diamA = graph_diameter(graphA)
diamB = graph_diameter(graphB)
diam_scale = max(diamA, diamB)
norm_metrics[k + "_norm"] = metrics[k] / diam_scale
```

Normalization prevents large networks from always having larger metric values

Other UMLs

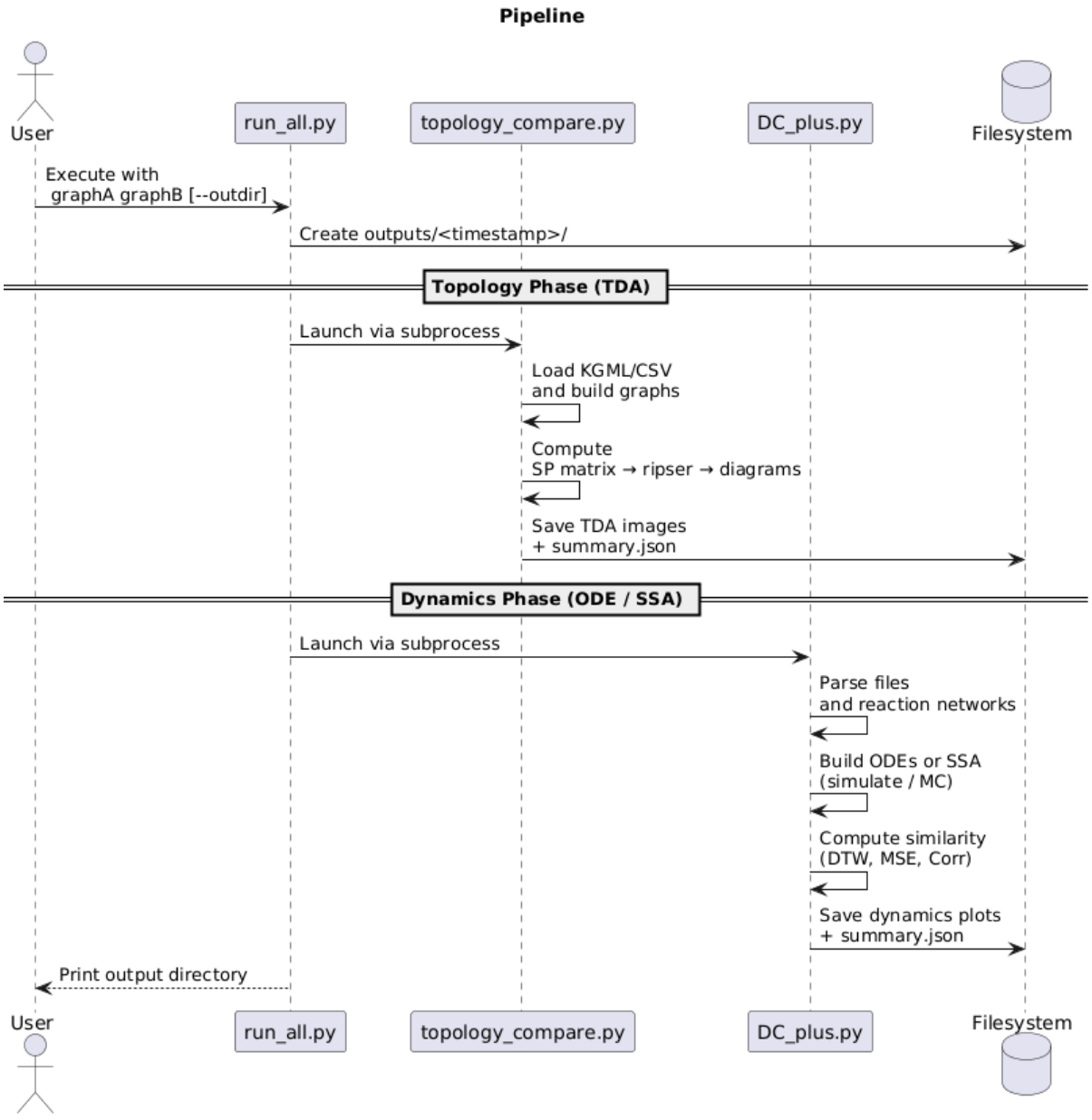

Figure 3: UML Diagram for Pipeline

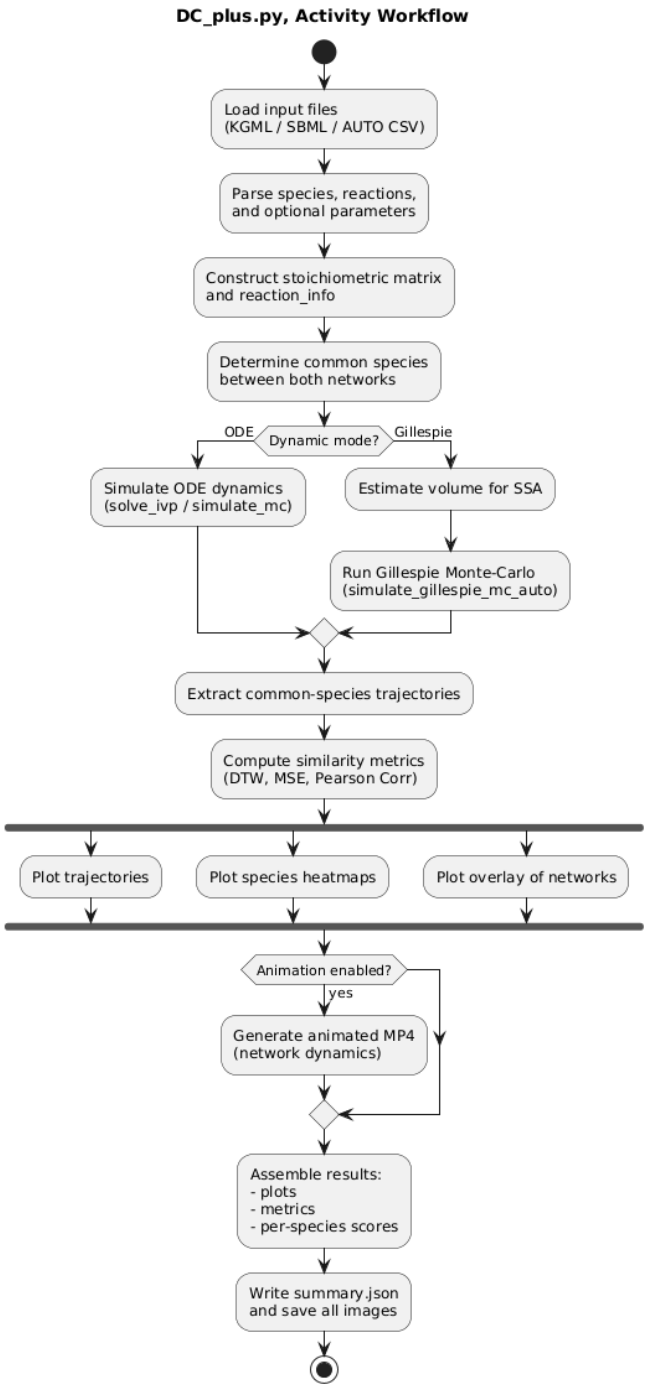

Figure 4: UML Diagram for DCPlus.py activity

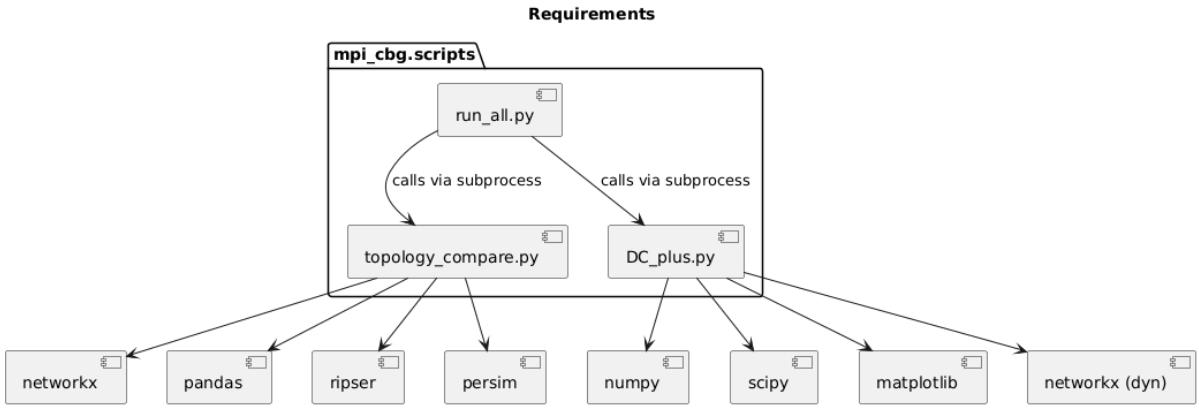
Requirements
mpi_cbg.scripts
run_all.py
calls via subprocess
calls via subprocess
topology_compare.py
DC_plus.py
networkx
pandas
ripser
persim
numpy
scipy
matplotlib
networkx (dyn)

Figure 5